Lecture 11: Tomasulo’s Algorithm; Branch Handling and Prediction
Tomasulo’s Algorithm

- Tomasolo, R.M.: “"An Efficient Algorithm for Exploiting Multiple Arithmetic Units"
- IBM 360/91 (three year after CDC 6600 and just before caches)
- CDB: Common Data bus – allow pending operand to be loaded simultaneously!
The Architecture

- 3 Adders
- 2 Multipliers
- Load buffers (6)
- Store buffers (3)
- FP Queue
- FP registers
- CDB: Common Data Bus
The Basic Sketch of the Steps

- **Issue:**
  - issue if an empty reservation station exists in the corresponding FU, and fetch operand if they are in Rs, otherwise --- a tag
  - otherwise stall and wait for a reservation slot

- **EX:** an inst at a reservation station with a pending operand will monitor CDB bus; when both operands are available – execute it!

- **WB:** when result is available, write it to CDB and any FU which waits this result.
Note that:
  – when an inst enters into a reservation station, the source operands are either filled with “values” or “renamed”
  – The new names are 1–to–1 correspondent to the FU names

Question:
How the output dependence is resolved?
That is when there are two pending writes to a register, how would a subsequent read get the correct (most recent) one, if the two write completed out–of–order?
Feature of Tomasulo Algorithm
(Compare with Scoreboard)

- The value of an operand (for any inst already issued in a reservation station) will be read from CDB. it will not be read from the reg. field.
- Instructions can be issued without even the operands produced (but know they are coming from CDB) or RAW does not cause stall.
Advantage of Tomasulo Alg.  
(with respect to scoreboard)

- Distribution of the hazard detection logic (due to distribution of reservation station) and CDB
- Stall due to WAW and WAR hazards are eliminated by R–renaming using reservation stations
Advantage:
- eliminating WAW, WAR hazards by reg. renaming
- out-of-order inst. execution and WAW, WAR elimination

Disadvantage:
- Hardware cost: high-speed associative M for tags + complex control logic
- one single CDB may be a bottleneck, while multiple CDB may be too costly (all associative – M must be duplicated)

Conclusion
- good for pipelined arch. which is difficult to schedule code and is short of regs
- future:
  - software scheduling reaches its end
  - we may see hybrid
    - static schedule of R–Rs
    - dynamic schedule load/store
Reorder buffer and Tomasolo Algorithm

- Many elements of Tomasulo’s algorithm are already included
- Major difference?

How WAW is handled?

In Tomasulo: this is by keeping a “tag” with each register $x$ and the tag is updated, each time a

$$x \leftarrow "+"$$

is issued, i.e. $x$-tag $\leftarrow "+3"$ means 3rd + unit is reserved.

when write back via CDB, the tag of FU is compare with tag of R:

if

$$\text{tag of FU} = x\text{-tag},$$

overwritten the R (e.g. X)

else

ignore the result
"The major challenge for these machines is to try to exploit large amounts of instruction–level parallelism. When the parallelism comes from unrolling simple loops, the original loop probably could have been run efficiently on a vector machine (see the next chapter). It is not clear that a VLIW is preferred over a vector machine for such applications; the costs are similar, and the vector machine is typically the same speed or faster. The open question in 1990 is whether there are large classes of applications that are not suitable for vector machines, but still offer enough parallelism to justify the VLIW approach rather than a simpler one, such as a superscalar machine."

– Patterson & Hennessy
Branch Handling

- The Problem
- Static Branch Prediction
- Dynamic Branch Prediction
- Architecture Support
Branch Behavior of Programs

- Branch frequencies can be significant for the so-called “integer code” (see Patt&Henn95, Fig. 3.24)
- Branch outcomes are usually not “random,” and some are more regular than others
Branch Behavior of Programs
Static Branch Prediction

- Prediction is done at compile time
- And it stays the same during the entire execution
Basic Static branch Prediction Techniques

- Simple (do not predict — so freeze the pipeline during a branch)
- Branch–not–taken
  (you do not know the branch target anyway! Cost of back–up!)
- Branch–taken
- Delayed branch
  (Compiler must fill the delay slots)
- Canceling/nullifying branch:
  Each branch instruction is encoded with a predicted "direction";
  if the real direction is "wrong" — the instruction at the delay slot is nullified!
Scheduling of the Branch–delay Slot
Improve Static Branch Prediction Accuracy

- Static(profile) Analysis –
  statically estimate with program analysis
- OP code heuristic
  ($\leq 0$, $< 0$, $= \text{const}$) will fail
- Pointer heuristic
  (test for null) will fail
- ...
- Loop header heuristic
- Call heuristic (not call)
- Store heuristic (not store)
- Return heuristic (not return)

Profile Report – dynamically counting events
What to do on a Mis–Predicted Branch?

- Pursue only predicted path and backup when it is wrong!
- Allow both arms proceed simultaneously and invalidate one when the outcome is known.

Need doubled resources for each branch (I-cache port, function units...)
- typically need to accommodate 4 branches : 8–16 resources.
Dynamic Branch Prediction

- Guess target dynamically and proceed marking result “tentative” means they cannot overwrite user accessible registers
Branch Prediction

- Success rate??
  - program behavior dependent
  - end of loop test
  - branch if (>0, <0) which one is regular?
- How to detect and guess?

BNE or not

Jump

```
Loop .
 .
 .
 Loop BNE .
 .
 .
 BNE Loop
 .
 .
 .
 Loop BNE x+1
 .
 .
 x Branch Loop
 .
 .
 x+1
 .
 .
```
• Guess is better than “do nothing” ~ careful

• Even random has 50% chance to win!

• A reasonable & effective way

• Validation through program traces
Study Method

- No statistically accepted model to characterize program behavior
- Trace–driven simulation
Loop buffer

All inst of a loop could be fetched from the buffer
used in CDC 6600, 7600, ......
Branch–Prediction Buffer (BPB)

- Do this at ID stage?
- A small buffer indexed by the instruction addresses (low order bits)
- 1–bit scheme
- 2–bit scheme
1–bit scheme

The memory location contains a bit indicating whether the branch was recently taken or not (how about almost always taken but not taken once?) – pay twice the price! of Misprediction.
The states in a two-bit prediction scheme. By using two bits rather than one, a branch that strongly favors taken or not taken – as many branches do – will be mis-predicted only once. The two bits are used to encode the four states in the system,
J.K.F. Lee and A.J. Smith:
“Branch Prediction Strategy and Branch Target Buffer Design”
IEEE Computer, Jan. 1994
Prediction Rate of a 4K Entry Prediction Buffer for SPEC Benchmarks
Note:

(1) Simple BPB is not useful when the target PC address is known at the same time branch result is known (e.g., DLX)

(2) BPB 2-bit has an accuracy of roughly 99% ~ 82% for SPEC benchmark with 4K entries.
Implementation of BPB

- As a special cache attached to the ID stage
- Or attached to each line of an inst cache
Correlation & Two–Level Branch Prediction

- A motivating example
  - Example 1

- Correlation predictor:
  - Example 2 (1–bit)

- Two–Level branch prediction
Example 1

b1: if (aa == 2) then
    aa == 0;
b2: if (bb == 2)
    bb == 0;
b3: if (aa ! == bb)
    { .......... }

behavior of b3 is correlated with b1 and b2
for example:
    b1 = T   b2 = T   b3 = F
Example 2

b1: if (d $\neq$ 0)
d =1;
b2: if (d $\leq$ 1)
{ .......... }

Obviously : b2 is correlated with b1. for example:
if  d $\geq$ 0 $\Rightarrow$ b1 = T $\Rightarrow$ b2 = F

Question:
assuming  d: 2,0,2,0,2,0,...
(1) for a 1–bit predictor with initial not taken then all branches are Mispredicted for both b1 and b2!
(2) but for a 1–bit correlation predictor with initial taken : all branches are correctly predicted with only the first d=2 wrong!
Branch Correlation

- Use of a branch history buffer
- Example: a 1–bit history buffer — it encodes the last prediction (taken or not taken), and based on this one of two predictors is chosen for use of the next prediction
- Note that the predictors themselves could be 1–bit, 2–bits, etc.
- We usually write (0,1), (1,1), (1,2), etc. such that
1-bit Correlation Scheme
<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>new b1 prediction</th>
<th>b2 action</th>
<th>new b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

(NO CORRELATION)
Two–Level Branch Prediction

- a (1,1) predictor:
  use only the last branch to select a pair of 2 x 1–bit predictor

- a (2,2) predictor
  use 2 bit (correlation of the last 4 branches) to select one of the 4 x 2–bit predictors.
A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.
Note:
Each predictor is in turn a two-bit predictor for that particular branch. The branch-prediction buffer shown here has a total of 64 entries; the branch address is used to choose four of these entries and the global history is used to choose one of the four. The two-bit global history can be implemented as a shifter register that simply shifts in the behavior of a branch as soon as it is known.
Comparison of two–bit predictors.

Frequency of Misprediction
Note:

A noncorrelating predictor for 4096 entries of 2-bit predictors are first, followed by a noncorrelating two-bit predictor with unlimited entries and a two-bit predictor with two bits of global history and a total of 1024 entries.